FlightLinux Project
Lessons Learned Report

Patrick H. Stakem
QSS Group, Inc.
June 15, 2002

Introduction

The intent of this report is to document the lessons learned during the conduct of the FlightLinux Project. Certain material from other project reports is repeated here, to make this report stand-alone. The Project website is located at http://FlightLinux.gsfc.nasa.gov, and full text of all of the project reports can be found there.

This work was conducted under task NAS5-99124-297, with funding by the NASA Advanced Information Systems Technology (AIST) Program, NRA-99-OES-08. The work is conducted by personnel of QSS Group, Inc. in partnership with NASA/GSFC code 586 (Science Data Systems), code 582 (Flight Software) and code 588 (Advanced Architecture & Automation). Work continued on this task under contract NAS5-99124-564, and concludes on June 30, 2002.

Executive Summary

The following section provides a synopsis of the lessons learned from this effort. Further detail is given in the body of the report.

1. **Developing software for an embedded system is always harder than you imagine, even when you allow for past lessons learned**. The UoSat-12 computer was a custom design by Surrey Space Technology Laboratories (SSTL). It is based on the now-discontinued Intel 80386EX processor, and is designed to be very close to a pc architecture. However, it does not have a BIOS (Basic Input Output System), a firmware based part of the operating system), and uses triple modular redundancy (TMR) memory organization. We were unable to located any applicable debugging hardware or software at GSFC after extensive search. We were also unable to locate any specific COTS hardware or software for debugging still available. Deploy development and debugging tools as early as possible. This allows the development and test team to get familiar with the tools and develop a toolbox of applications. Remote access to the test system saves time. The test system and development systems can be arbitrarily located. Don't re-invent the wheel. With the Linux system, almost anything you can imagine already exists as code on the web.

2. **Exporting Satellite Control Software is a big deal.** As we abruptly learned about 1 year into the project, the International Trafficking in Arms (ITAR) regulations apply to any satellite control software, of which FlightLinux, as the onboard operating system, would be a part. ITAR regulations would require a review and certification by Goddard Space Flight Center (GSFC), National Aeronautics and Space Administration (NASA),

Department of Defense (DoD), Department of State (DOS), and other agencies, a very time-consuming process.

3. **Outreach works**. Our main approach to public outreach has been the project website, which has been up since the beginning. Since we posted our goals of keeping the FlightLinux open source, within the meaning of the gnu license (www.gnu.org/license), we have had numerous offers of collaboration on the project. These include representatives of worldwide aerospace companies, and individuals. The interest in the FlightLinux Project is growing, due to increasing exposure of the website. We are aware of a dozen flight projects using Linux. Many individuals contacted us for additional information, or to volunteer to work on the project. We declined these offers, as we did not have an approach to include their efforts (see item 5). However, the level of interest in this project was very high, and world-wide, attracting very high levels of talent to the possibility of contributing to a space-based application. Most notably, Richard Stallman, the originator of the Free Software movement, conducted an extended dialog with us. A recent Internet search found over 800 references to the FlightLinux Project, in 23 countries. Examples of the results of the Outreach effort will be included in the final report.

4. **Partnering and collaborating works**. The work was conducted by a team from QSS Group, Inc., in conjunction with NASA/GSFC Codes 586 (Science Data Systems), 582 (Flight Software), and 588 (Advanced Architectures and Automation). The government-industry team approach worked smoothly, and we cooperating and collaborated with various projects, most notably, the *Orbiting Missions as Nodes on the Internet* (OMNI) Project in Code 588. By the nature of the mutual interests, code-sharing within the spirit of the gnu public license was possible. Resource sharing also was commonplace. The OMNI Project hosted the UoSat-12 breadboard that we used for development and test. It is possible for a government-industry team to work together successfully towards common goals, with diminished parochialism and territorial behavior.

5. **An approach is needed to allow and manage Open Source software development for Mission Critical NASA Projects**. The FlightLinux Project explored new issues in the use of "free software" and open-source code, in a mission critical application. Open-source code, as an alternative to proprietary software has advantages and disadvantages. The chief advantage is the availability of the source code, with which a competent programming team can develop and debug applications, even those with tricky timing relationships. The Open-Source code available today for Linux supports international and ad hoc standards. The use of a standards-based architecture has been shown to facilitate functional integration.

Many issues on the development and use of Open-Source software on government-funded and mission-critical applications are still to be explored. Debates continue in the media about the relative merits of Open Source versus proprietary software with respect to security issues. This is an area that is both timely and critical, and will need to result in a definition of policy and preferred approach, as well as guidelines to projects.

Background

The FlightLinux project had the stated goal of providing an on-orbit flight demonstration of the Linux software, resulting in a Technology Readiness Level of 7. The proposed platform was the WIRE spacecraft. After extensive discussions with the WIRE Project and software developers, this approach was abandoned due to the complexity involved. We then explored a spectrum of options, including the Surrey Space Technology (SSTL) SNAP, and potential GSFC near-term missions. We needed to have a mission that was far enough along to allow us to meet our timetable for on-orbit testing, but also one that was amenable to hosting our code. We were able to get access to the SSTL UoSat-12 spacecraft, already in-orbit. The OMNI project of Code 588 at GSFC has procured a breadboard of the Surrey UoSat-12 OBC, that is being used for testing. In addition, telecommunications facilities at Building 23 at GSFC will allow direct communication with the UoSat-12 spacecraft.

Because almost all of the effort in developing onboard computer hardware for spacecraft involves adapting existing commercial designs, the logical next step is to adapt commercial off-the-shelf (COTS) software, such as the Linux operating system. Given Linux, many avenues and opportunities become available. Web serving and file transfers become standard features. Onboard local area networks (LAN's) and an onboard file system become "givens." The Java language becomes trivial to implement. Commonality with ground environments allows rapid migration of algorithms from ground-based to the flight system, and tapping into the worldwide expertise of Linux developments provides a large pool of talent. Full source for the operating system and drivers is available on day one of the project.

Since we posted our goals of keeping the FlightLinux open source, within the meaning of the gnu license, we have had numerous offers of collaboration on the project. These include representatives of worldwide aerospace companies, and individuals. The interest in the FlightLinux Project is growing, due to increasing exposure of the website. We are aware of a dozen flight projects using Linux. These are documented later in this report.

We have defined the steps to a space-flight demonstration of the Linux operating system. Regardless of the implementation architecture, certain pivotal issues must be defined. This will be done in a series of reports. These reference reports will be archived together in one place along with ongoing research related to the topics. The key issues include: the architecture of the target systems, the nature of application software, the architecture of an onboard LAN, and the requirements for support, the architecture of the onboard storage system, the requirements for support, and the nature and design of the software development testbed.

The Target Architecture Technical Report examines the current, near-term, and projected computer architectures that will be used on board spacecraft. The resulting list allows examination of the feasibility and availability of Linux. The choice of the actual architecture for implementation will be determined more by opportunity of a flight than by choice of the easiest or most optimum architecture.

The *POSIX Report* examines and documents the POSIX-compliant aspects of Linux and other Flight Operating systems as well as the POSIX-compliant nature of legacy flight application software. This is an ongoing effort by GSFC Code 582.

The Onboard LAN Architecture Report discusses: 1) the physical level interfaces on existing and emerging missions and 2) the device drivers required to support IP over these interfaces. Ongoing work in this area is being done by the Consultive Committee on Space Data Systems (CCSDS) and the OMNI Project (GSFC, Code 588). The choice of a demonstration flight will define which interfaces will need to be implemented first. In addition, those interfaces with COTS drivers, and those for which device drivers need to be defined will be delineated.

The Bulk Memory Device Driver Report will define the approach to be taken to implement the Linux file system in the bulk memory ("tape recorder") of the spacecraft onboard computer. It will define which elements are COTS and which need to be developed.

These reports are living documents and will be updated to document new developments. The reports will be stand-alone, but will reference the other reports as required. A major purpose of the reports will be to collect in one area the COTS aspects of the specific aspect of the FlightLinux implementation so that attention may be focused on the remaining "missing pieces." A summary of each report is presented below.

Target Architectures

Various microprocessor architectures have been and are being adapted from commercial products for space flight use. For all of the primary architectural candidates we identified, Linux is available in COTS form. The primary hardware for flight computers in the near term are derived from the Motorola PowerPC family (RHPPC, RAD6000, RAD750), the SPARC family (EH32), the MIPS family (Mongoose, RH32), the Intel architecture (space flight versions of 80386, 80486, Pentium, Pentium-II, Pentium-III), and the Intel ARM architecture. Versions of FlightLinux for the PowerPC and MIPS family are important goals. Because almost all of the effort in developing onboard computers for spacecraft seems to involve adapting existing commercial designs, the logical next step is to adapt COTS software, such as the Linux operating system.

Given the candidate processors identified in missions under development and planned in the short term, we then examined the feasibility of Linux ports for these architectures. In every case, a Linux port was not only feasible, but is probably available COTS. Each needs to be customized to run on the specific hardware architecture configuration of the target board.

Existing space processors in recent or planned use include the RAD6000, the RH32, and the MIPS-derived Mongoose-V. Generally, Linux requires a Memory Management Unit (MMU) for page-level protection, as well as dynamic memory allocation. However, ports of Linux (uCLinux) exist for the Motorola ColdFire processor series and similar architectures, all without memory management. The Mongoose architecture does not include memory management hardware. A Mongoose port of Linux is feasible, and this has been examined in conjunction with GSFC, Code 582, Flight Software Branch. The future usage plans of these hardware architectures determined the direction of our efforts on the FlightLinux software ports.

Emerging space processors include Honeywell's RHPPC, the Lockheed's RAD750, ESA's ERC32, and the Sandia Lab's radiation-hard Pentium. All are viable targets for FlightLinux. The RHPPC and the RAD750 are variations of the Motorola PowerPC architecture. GSFC Code 586 already has Linux running on the PowerPC architecture, in a laboratory environment. The Intel (Pentium) version of Linux is the most common, and can be found in the Code 586 lab as well. ESA's ERC32 is a variation on the SPARC architecture, and Linux is available for the Sun Sparc architecture. The term COTS in this context should be taken to mean that a commercial version for that processor architecture is available. A specific port for the Flight Computer embedded board involves coding specific device drivers, reconfiguration, and recompilation of the kernel. Linux is a 32-bit operating system, appropriate for matching the emerging 32-bit class of flight computers. The results of the target architecture study are given in ref [1].

POSIX

POSIX is an IEEE standard for a Portable Operating System based Unix. The use of a POSIX-compliant operating system and applications has many benefits for flight software. Among these benefits are 1) software library reuse between missions and 2) software commonality between ground and flight platforms. For compliant code, the function calls, arguments, and resultant functionality are the same from one operating system to another. Source code does not have to be rewritten to port to another environment. Linux variants are mostly, but not completely, POSIX-compliant. The POSIX standards are now maintained by an arm of the IEEE called the Portable Applications Standards Committee (PASC).

The advantages of Linux are numerous, but the requirements for spacecraft flight software are unique and non-forgiving. Traditional spacecraft onboard software has evolved from being monolithic (without a separable operating system), to using a custom operation system developed from scratch, to using a commercial embedded operating system such as VRTX or VxWorks. None of these approaches have proved ideal. In many cases, the problems involved in the spacecraft environment require access to the source code to debug. This becomes an issue with commercial vendors. Cost is also an issue. When source code is needed for a proprietary operating system, if the manufacturer chooses to release it at all, it is under a very restrictive non-disclosure agreement, and at additional cost. The Linux source is freely available to the team at the beginning of the effort.

As a variation of Linux, and thus Unix, FlightLinux is Open Source, meaning the source code is readily available and free. FlightLinux currently addresses soft real-time requirements and is being extended to address hard real-time requirements for applications such as attitude control. There is a worldwide experience base in writing Linux code that is available to tap.

The use of the FlightLinux operating system simplifies several previously difficult areas in spacecraft onboard software. For example, the FlightLinux system imposes a file system on onboard data storage resources. In the best case, Earth-based support personnel and experimenters may network-mount onboard storage resources to their local file systems. The FlightLinux system both provides a path to migrate applications onboard and enforces a commonality between ground-based and space-based resources.

We pursued the IEEE POSIX compliance issues of standard embedded Linux, in parallel with an effort in GSFC Code 582, which has collected a library of POSIX-compliant flight applications software. FlightLinux also enables the implementation of the Java Virtual Machine, allowing for the up-link of Java applets to the spacecraft.

Linux is not by nature or design a real-time operating system. Spacecraft embedded flight software needs a real-time environment in most cases. However, there are shades of real time, specified by upper limits on interrupt response time and interrupt latency. We can generally collect these into hard real-time and soft real-time categories. Examples of hard real-time requirements are those of attitude control, spacecraft clock maintenance, and telemetry formatting. Examples of soft real-time requirements include thermal control, data logging, and bulk memory scrubbing.

Unix, and Linux, were not designed as real-time operating systems, but do support multi-tasking. Modifications or extensions to support and enforce process prioritization are necessary to apply Linux to the embedded real-time control world.

Many real-time schedulers for Linux are available for download. These are a Rate Monotonic Scheduler, which treats tasks with a shorter period as tasks with a higher priority, and an Earliest Deadline First (EDF) scheduler. Other approaches are also possible. It is not clear which approach provides the best approach in the spacecraft-operating environment. The results of our POSIX investigations of the Linux system and legacy flight application code are documented in Ref. [2].

The Bulk Memory Device Driver

Spacecraft onboard computers do not usually employ rotating magnetic memory for secondary storage. Initially, magnetic tape was used, but now the state of the art is to use large arrays of bulk Dynamic Random Access Memory (DRAM), with various error detection and correction hardware and/or software applied.

The current state of the art of spacecraft secondary storage is bulk memory, essentially large blocks of DRAM. This memory, usually still treated as a sequential access device, is mostly used to hold telemetry during periods when ground contact is precluded. Bulk memory is susceptible to errors on read and write, especially in the space environment, and needs multi-layer protection such as triple-modular redundancy (TMR), horizontal and vertical Cyclic Redundancy Codes (CRC), Error Correcting Codes (ECC), and scrubbing. Scrubbing can be done by hardware or software in the background. The other techniques are usually implemented in hardware. With a MMU, even using 1:1 mapping of virtual to physical addresses, the MMU can be used to re-map around failed sections of memory.

Although we usually think of bulk memory as a secondary storage device with sequential access, it may be implemented as random access memory within the computer's address space. This is the case with UoSat-12.

The Flash File System (FFS) has been developed for Linux to treat collections of flash memory as a disk drive, with an imposed file system. Although we are dealing with DRAM and not flash, we can still gain valuable insight from the FFS implementation. In addition, the implementation of Linux support for the personal computer memory card international association (pcmcia) devices provides a useful model.

The onboard computer on the UoSat-12 spacecraft has 128 megabytes of DRAM bulk memory. It is divided into four banks of 32 megabytes each, mapped through a window at the upper end of the processor's address space. This is the specific device driver that the QSS team develops and uses as a model for future development of similar modules. The current software of the UoSat-12 onboard computer treats this bulk memory as paged random access memory and applied a scrubbing algorithm to counter environmentally induced errors.

The ram disk is a disk-like block device implemented in RAM. This is the correct model for using the bulk memory of the onboard computer as a file system. Multiple RAM disks may be allocated in Linux. The standard Linux utility "mke2fs," which creates a Linux second extended file system, works with RAM disk, and supports redundant arrays of inexpensive disks (RAID) level 0. We implemented this on our test machine.

This initial version of the RAID driver uses memory mirroring, with memory scrubbing techniques applied. In the simplest case, we treat three of the four available 32-megabyte memory pages as a mirrored system. The memory scrubbing technique is derived from the current scheme used by SSTL, as is the paging scheme. The next version of the driver uses all four of the available 32-megabyte memory pages with distributed parity. The performance with respect to write speed is expected to be less than with the Level 0, but the memory resilience with respect to errors is expected to be much better.

It is unclear without further testing whether the RAID technique will be sufficient to counter the environmentally-induced errors expected in the bulk memory on-orbit. It is generally accepted that RAID is not intended to counter data corruption on the media, but rather to allow data recovery in case of media failure. A defined testing approach will be used with the bulk memory device driver on the breadboard facility. More extensive testing on-orbit with the UoSat-12 spacecraft is required to validate the approach. The Bulk Memory Device driver approach that we defined is documented in Ref. [4].

Onboard LAN

Given that the Linux operating system is onboard the spacecraft, support for a spacecraft LAN becomes relatively easy. Extending the onboard LAN to other spacecraft units in a constellation also becomes feasible, as does having the spacecraft operate as an Internet node.

For space to ground communication, FlightLinux can build on the IP-in-space work validated by GSFC Code 588's OMNI Project. Interface between spacecraft components is usually provided by point-to-point connections, or a master/slave bus architecture. The use of a LAN onboard is not yet common. This is partially due to the lack of space-qualified components.

A LAN-type architecture is typically used in office and enterprise environments (and spacecraft control centers). It provides a connection between peer units, or clients and servers. The typical LAN uses a coax or twisted pair connection at a transmission rate of 10 megabits per second, a twisted pair connection at 100 megabits per second, or optical at 155 megabits per second, with higher speeds possible.

Usually, a LAN is configured with a repeating hub or a central switch between units. The standard protocol imposed on the physical interface is Transmission Control Protocol /Internet Protocol (TCP/IP), although others are possible (even simultaneously). The TCP/IP protocol has become a favored approach to linking computers around the world. Linux and most other operating environments support the protocol.

The UoSat-12 configuration allows us to exercise the TCP/IP and Controller area network (CAN) bus components of an onboard LAN. Evolving physical layer interfaces for use onboard the spacecraft include 100 megabit Ethernet and Firewire (IEEE-1394). Although 1553 device drivers exist for Linux, they only allow the use of the legacy bus in the classical master-slave architecture. Ongoing work by the Spacecraft Onboard Interfaces (SOIF) group within CCSDS is defining the application of IP-over-1553, and, more generally, IP over a master-slave architecture. The results of our Onboard LAN architecture investigations are documented in Ref. [3].

Code Integration and Testing

An initial build of the FlightLinux software has been running since March 2001. The UoSat-12 breadboard has been available at the OMNI Lab since April 2001. The breadboard has an associated Windows-NT machine to load software via the CAN bus or the asynchronous port and to provide debugging visibility. From our facility at QSS, we can access the breadboard facility via the "PC-Anywhere" software with the appropriate link security. The following sections give background information on some of the system components.

80386EX Processors

The 80386EX model includes the memory management features of the baseline 80386, along with an interrupt controller, a watchdog timer, synchronous/asynchronous serial I/O, DMA control, parallel I/O and dynamic memory refresh control. These devices are DOS-compatible in the sense that their I/O addresses, direct memory access (dma) and interrupt assignments correspond with an IBM PC board-level architecture. The DMA controller is, however, an enhanced superset of the Intel 8237A DMA controller. The 80386EX processor core is static meaning that the clock can be slowed or stopped withour loss of state.

The 80386EX includes two DMA channels, three channels of Intel 8254 compatible timer/counter, dual Intel 8259A interrupt controller functionality, a full-duplex synchronous serial I/O channel, two channels of Intel 8250A asynchronous serial I/O, a watchdog timer, 24 lines of parallel I/O, and support for dram refresh. The 80386EX can interface with the 80387 math co-processor, and the SSTL breadboard is equipped with an 80387SL.

Specific Device Drivers

A large number of device drives for custom I/O interfaces are available in open-source form for Linux. The physical-level I/O interfaces most likely to be required include asynchronous serial, synchronous serial, 1553/1773, CAN (ISO 11898), and Ethernet (IEEE 802.3 10-base-T). The device driver needs to be customized for the specific physical layer hardware used to implement the interface. We have identified existing software drivers for all of these. The UoSat-12 OBC uses asynchronous serial (debug only) synchronous serial, 10Base-T, and CAN. CAN and 10Base-T drivers are COTS, and the OMNI group has developed a synchronous serial HDLC device driver that we can use.

FlightLinux-specific Kernel Enhancements or Application Code

Certain extensions to a standard Linux kernel, beyond an embedded version, will have to be made for the unique space flight environment. These enhancements may take the form of application code, running under the kernel; this issue is currently "to be decided." Such enhancements will include a bulk memory device driver/file system (using bulk memory as a file system - to be discussed in a subsequent report), a memory scrub routine to periodically check and correct memory for radiation-induced errors, and a watchdog timer reset routine, to detect radiation-induced latchup of the processor.

Non-BIOS Systems

The UoSat-12 80386EX embedded board does not contain a BIOS, the firmware-based basic I/O system that is common in desktop computers. The functions that a BIOS provides include power-on self test, hardware configuration establishment, and software environment establishment. The UoSat-12 board does have a 32-kilobyte ROM that contains the Loader code. Two versions are available: one for loading from the async serial maintenance port and another for loading via the CAN bus. The loader code includes an Initialization routine, which provides some of the functionality of the BIOS in the sense of configuring the hardware and software environment. Run time support to programs is not provided. After the code is loaded, the ROM is mapped out of the memory space.

The Linux system in general, at least the 80x86-based implementations, rely on a BIOS for the initial system load. When a BIOS is not present, as in an embedded system, the functionality must be provided by other means. Once the Linux kernel is up and running, Linux does not rely on the BIOS. Architectures other than the 80x86 contain firmware that may be broadly classified as a "BIOS."

Embedded Debugging Tools

Because the embedded system does not have the usual human interfaces such as keyboard and screen, alternative approaches must be found for debugging. One of these is the gnu debugger (gdb). When the target system (in this case, the SSTL 80386EX board) does not have a console, the gdb will run remotely on the host, with several minimal modules in the target and a link via the serial port. We implemented the gdb on the UoSat-12 breadboard. UoSat-12 Breadboard debugging options we explored include:

1. BlueCat Vendor Support - We started with the idea of using the BlueCat release of Linux from LynuxWorks, and we have copies of Version 2 and 3. We are currently using the ELKS distribution of Linux, because it is very simplified, and can be brought up, for example in real mode on the 80386; it does not require protected mode, with its associated complexities. The idea was to get something working with ELKS, then switch to BlueCat. One approach is to upgrade to the latest (version 4) release of BlueCat Linux, and get the associated priority support option. This would make available to us a BlueCat application engineer. The cost of this option was not within the budget. The Omni lab also currently uses BlueCat, but does not have a copy of Version 4. BlueCatRT - the real time extension, was announced on June 10, 2002.

2. ICE - an in-circuit emulator, uses a pod to replace the cpu, that cables to a box. A 80386EX ICE unit was not available at GSFC for loan. They are very expensive pieces of equipment. This turns out not to be an option with the UoSat-12 breadboard, as the cpu

chip is soldered in. The UoSat unit is more of a proto-flight unit than a breadboard. It is not conformally coated.

3. Logic Analyzer - There is a logic analyzer available in the Omni Lab. This unit can be considered a generic ICE - it can capture and display logic signals, can be triggered by predefined events, and can buffer a series of states on logic line. It would normally be used on the cpu's address and data bus, and some selected control signals. It is not processor-specific. There is a learning curve in its use, and the operator must understand software-hardware interactions.

4. Rom-based Monitor. On the UoSat-12 board, the ROM based monitor contains the load and dump code, and rudimentary hardware set-up routines. It is NOT a BIOS. We have the ability to plug a ROM emulator into the ROM socket, or to program custom proms. This would require an external prom programmer unit. A much better solution is to have the firmware in a flash-rom, which can then be modified in-circuit. But, this option has to be included in the board design, and is not a feature of the UoSat-12 OBC. With a large enough flash rom, significant portions of the Linux kernel could be included as well. This is the approach to take if the hardware can be specified.

5. A rom emulator plugs into the rom's socket, replacing the rom. This is also sometimes referred to as a Prom-ICE. Since the prom is replaced by ram on the debugging system, many options are possible. We did not located such a unit for the UoSat.

The Embedded Testbed Report (ref. 5) gives additional details on these facilities.

The initial FlightLinux software load is approximately 400,000 bytes in size. The nature of the UoSat-12 memory architecture at boot time limits the load size to less than 512,000 bytes. After the loader, which is read-only memory (ROM)-based, completes, 4 megabytes of memory is available. The software load includes: 1) a routine to setup the environment and 2) a routine to decompress and start the Linux kernel. The kernel is the central portion of the operating system, a monolithic code entry. It controls process management, input/output, the file system, and other features. It provides an executive environment to the application programs, independent of the hardware.

Extensive customization of the SETUP routine, written in assembly language, was required. This routine in its original form relies on BIOS (Basic Input/Output System) calls to discover and configure hardware. In the UoSat-12 configuration, there is no BIOS function, so these sections were replaced with the appropriate code. Sections of SSTL code were added to configure the unique hardware of the UoSat-12 computer. The SETUP routine then configures the processor for entry to Protected Mode and invokes the decompression routine for the kernel. The SETUP routine is approximately 750 bytes in length and represents the custom portion of the code for the UoSat-12 software port. This product is usually referred to as the Board Support Package. The remaining code is COTS Linux software. This process is the same for any FlightLinux port.

We modified the standard Linux SETUP routine, written in assembly language, to be Table-driven. This had the added advantage of addressing the export restriction issues. The breadboard architecture includes an asynchronous serial port for debugging. We used this extensively for debugging the SETUP module. On the spacecraft, the asynchronous port exists, but it is not connected to any additional hardware.

FlightLinux was implemented in an incremental manner. The initial software build does a "Hello, World" aliveness indication via the asynchronous port and allows login. The synchronous serial drivers must be integrated to allow communication in the flight configuration. The bulk memory device driver, which uses the 32-megabyte modules of extended memory as a file system, will be added next. The breadboard has a single 32-megabyte module, and there are four modules in the flight configuration. The CAN bus drivers and the network interface can be added later. Additional modules can be uplinked later on an incremental basis.

We did not complete the testing of the FlightLinux software on the UoSat-12 breadboard within the allocated resources. It is difficult to reasonably estimate the effort required to complete the testing.

TRL Assessment

This Project started with an estimated TRL level of 3. We progressed to a TRL level estimated at greater than 5. Our goal was TRL level 7. The steps required to progress from the current state to TRL Level 7 are as follows:

1. Debug the FlightLinux load on the UoSat breadboard, level of effort tbd

2. Validate the approach to software export. Level of effort, tbd.

3. Have SSTL download the software and test it on their breadboard facility. Jointly develop an on-orbit test plan and associated procedures. This should take several weeks.

4. Uplink the software to the UoSat-12 Spacecraft and verify the load. This should take about 1 day.

5. Execute the test plan. This should take about 2 weeks, although the testing period may be open ended, if the software is left in place.

Lessons Learned

1. **Developing software for an embedded system is always harder than you imagine, even when you allow for past lessons learned**. The UoSat-12 computer was a custom design by Surrey Space Technology Laboratories. It is based on the discontinued Intel 80386EX processor, and is designed to be very close to a pc architecture. However, it does not have a BIOS, a firmware based part of the operating system, and uses triple modular redundancy (TMR) memory organization. We were unable to located any applicable debugging hardware or software at GSFC after extensive search. We were also unable to locate any specific COTS hardware or software for debugging still available.

The normal boot process for the Linux system involves the BIOS invocation of the SETUP routine, which is coded in assembly language, and prepares data tables for the kernel. After SETUP finishes its job, control is transferred to the kernel code. In the UoSat-12 case, because there was no BIOS code per se, we used the SSTL loader to load the SETUP and kernel code, and transfer control to SETUP. The loader code, contained in a PROM on the UoSat-12 breadboard, did some hardware configuration, but did not include the full functionality of a BIOS.

After we debugged the SETUP routine, control was transferred into the Linux Kernel. At this point we were in the netherworld with no visibility of processes, before the kernel's serial console had not yet been configured. We could not make use of the gdb tool (gnu debugger), because it relies on having the kernel serial port available. We did not have a hardware cpu probe available, although the OMNI Project offered us the use of a logic analyzer. At the end of the available funds, we did not have the path through the kernel code working or debugged. This may have been due to a configuration parameter we missed in SETUP, or a memory configuration issue (an artifact of the loader). We employed extensive code reviews and traces, but were not able to isolate the problem within the resources available.

On the positive side, using the breadboard remotely at GSFC from our facility about a mile away worked very well. In worst case, we could drive over to the breadboard in a matter of minutes, but this was rarely necessary. It was much easier to use than if the facility had been located overseas.

If we had to start this project over with our current knowledge, we would more carefully assess the support systems available for the target architecture. We chose the UoSat-12 system more due to availability than a rigorous examination of suitability. To have a chance of achieving a Technology Readiness Level (TRL) of 7, on-orbit testing, we had to find a project that was already on-orbit, or close to launch. This severely limited our decision space. In retrospect, two years was probably too short for this project. Alternately, we might have chosen to stop at TRL level 5, and find a Flight project to collaborate with for further efforts.

2. **Exporting Satellite Control Software is a big deal.** As we abruptly learned about 1 year into the project, the International Trafficking in Arms (ITAR) regulations apply to any satellite control software, of which FlightLinux, as the onboard operating system, would be a part. ITAR regulations would require a review and certification by GSFC, NASA, DoD, DOS, and other agencies, a very time-consuming process.

Initially, we would have been faced with the review of perhaps a half-million lines of code for the Linux system. However, we realized that all of that was already in the public domain and available for download internationally. The only unique part would be the SETUP routine. (The configuration of the kernel, a key part of the FlightLinux load, involves the choice of standard modules and device drivers to include and exclude form the software build.). Given this, we reduced the unique problem to several hundred lines of code.

We modified the standard Linux SETUP routine, written in assembly language, to be table-driven. This had the advantage of addressing the export restriction issues, while making the SETUP routine more generic. Our proposed approach, which has not been validated by legal or export-control authorities, is that the generic version of the SETUP code, which we released back into the public domain under the terms of the gnu license, is not an issue. The contents of the table, which is data, not code, is the critical aspect.

We filed a form 1679, a NASA Disclosure of Invention and New Technology (including Software) with the GSFC Patent Counsel. On July 20, 2001, we answered the GSFC Global Concerns Statement. We also prepared the "GSFC Software Public Disclosure Export Control Checklist." It must be emphasized that our proposed approach to the export control issue has not been validated, and there may be further stumbling blocks to be encountered. However, other Linux based software, such as Beowulf, has been through the process successfully.

Reference: emails

Subject: Re: policy or position
Date: Thu, 17 May 2001 15:51:40 -0400
From: Lee Holcomb <lholcomb@hq.nasa.gov>
To: Pat Stakem <pstakem@qssmeds.com>

Pat:

Outstanding questions and efforts. I have in fact spoken to Ed Frankel our agency general council on this very topic. It was a result of the effort we had to go thru to get a release authority for the Beowulf software that caused the discussion with Code G. Last year the Presidents IT Advisory Council studied the same subject. I have a keen interest in developing an effective policy that allows the government to both use open source (which includes the GNU requirement of publication) and the requirement for export clearance. Let's discuss this more fully.

Lee
--
Lee Holcomb
NASA Chief Information Officer
(202) 358-1824

Subject: Re: policy or position
Date: Thu, 17 May 2001 17:01:37 -0400
From: Lee Holcomb <lholcomb@hq.nasa.gov>
To: Pat Stakem <pstakem@qssmeds.com>

Your points are mostly on target. The export issues is an issue because you are developing satellite computing capability which is on the State Department Munitions list. I realize that this is in direct conflict with the GNU policy. I believe that the government needs to take into account the free software issues. That said, the best answer is that we are formulating our policy. I belive we can give you a specific determination for your project. i suggest you and I begin a dialog on this issue and I will bring in general council.

Lee
--
Lee Holcomb
NASA Chief Information Officer
(202) 358-1824

Subject: Re: policy or position
Date Fri, 18 May 2001 16:07:51 -0400
From: Lee Holcomb <lholcomb@hq.nasa.gov>
To: Pat Stakem <pstakem@qssmeds.com>

Pat:

I have jumped into this issue with some comments, but not enough time to really work your issues to ground. I would suggest you work internal at GSFC to make sure that my comment on the need for an export lic is required. If you are operating on an government to government MOU, this may serve to allow the export in this unique case. Again, this is an issue for legal council and patent /export people at GSFC to advise you. I would be interested in the outcome.

Lee
--
Lee Holcomb
NASA Chief Information Officer
(202) 358-1824

3. **Outreach works**. Our main approach to public outreach has been the project website, which has been up since the beginning.

Since we posted our goals of keeping the FlightLinux open source, within the meaning of the GNU license, we have had numerous offers of collaboration on the project. These include representatives of worldwide aerospace companies, and individuals. The interest in the FlightLinux Project is growing, due to increasing exposure of the website. We are aware of a dozen flight projects using Linux. Many individuals contacted us for additional information, or to volunteer to work on the project. We declined these offers, as we did not have an approach to include their efforts (see item 5). However, the level of interest in this project was very high, and world-wide, attracting very high levels of talent to the possibility of contributing to a space-based application. Most notably, Richard Stallman, the innovator of Free Software, conducted an extended dialog with us. Examples of the results of the Outreach effort will be included in the final report.

Our website has been our primary Outreach tool. Here is a list of other Linux in Space Efforts we have corresponded with. In a sense, our evangelical efforts have paid off, and many projects are now considering and evaluating the use of Linux in the onboard environment.

A research Internet search found over 800 references to the FlightLinux Project in 23 countries (Russia, Spain, Mexico, Germany, UK, Italy, France, Greece, Argentina, Netherlands, Czech Republic, Finland, China, Taiwan, Poland, Denmark, Norway, Brazil, Hungary, Canada, Bulgaria, Romania, and Belgium).

Linux in Space efforts			June 2002	
Group	**Country**	**Mission**	**cpu**	**status**
GSFC	USA	ST-7	PPC-750	paper study
ASRI	Australia	JAESAT	ARM	in development
Hacettepe U.	Turkey	UPESAT	8086	in development
QinetiQ (DERA)	UK	STRV	ERC-32	in work
Montana State U.	USA	sounding rocket	Pentium	flight, 2004
ITT	USA	next gen s/c	PPC-750	Prototype in 2003
JPL	USA	Europa Orbiter	PPC-750	alt. to VxWorks
nsbf	Italy	balloon	80386	in work
Honeywell Space	USA	-?-	PPC-750	-?-
U. Michigan	USA	Mars Rover	-?-	prototype
Navy Postgrad	USA	various	8086	in work
U. NSW OMNI/GSFC	Australia US	BlueSat CANDOS	ARM 80686	In work STS-107

Notes:

1. The in-house ST-7 Study by GSFC chose Linux for the onboard operating system. In their words, "Real-time Linux and VX-Works were both examined as potential candidates for an Operating System (OS) for the ST7 spacecraft. While each OS offers its own unique set of advantages, Real-time Linux was chosen to support the ST7 Payloads for a number of reasons." Contact is Richard Schnurr, GSFC Code 560, 301-286-6069.

2. ASRI, contact is Geoffrey O'Callaghan, gocallag@au1.ibm.com. Student microsatellite project. http://www.asri.org.au/

3. Hacettepe U. contact Sefer Bora, bora@lisesivdin.net. Undergraduate Physics satellite project.

4. QuinetiQ - a spin-off of DERA in the U.K. contact is Malcolm Appleton, MAPPLETON@qinetiq.com. We have a MOU with QuinetiQ to port FlightLinux to their STRV1d engineering model.

5. Montana State U. contact is Charles Kankelborg, kankel@icarus.physics.montana.edu. This is a NASA funded sounding rocket. Their description of the project is " Flight from White Sands Missile Range is tentatively spring, 2004. We are planning an internal PDR for February 2002. We intend to fly a ruggedized Pentium single-board computer for control and data handling. We're expecting roughly half a gigabyte of data from our solar imaging spectrograph during a 5-minute data-taking window. Data acquisition is via one or two DMA cards (TBD), with a TBD interface to telemetry. All data will be stored to flash disk, and as much will be downlinked via telemetry as possible. Control functionality is limited to some simple commanding of cameras and shutters via serial lines."

6. ITT - contact is Chris Langford, Chris.Langford@itt.com

7. JPL - Europa Orbiter. Contacts are Kenny Meyer, Kenny.Meyer@jpl.nasa.gov, and Len Day, len.day@jpl.nasa.gov.

8. NSBF - Italian National Research Council, contact is Dr. Enzo Pascale, pascale@iroe.fi.cnr.it.

9. Honeywell Space, contact is Douglas Jerome, jerome@mate2000.az76.honeywell.com.

10. University of Michigan, Mars Rover Project, contact is Marius Aamodt Eriksen, marius@umich.edu.

11. Naval Postgraduate School - contact is Jim Horning, jahornin@nps.navy.mil.

12. University of New South Wales, Australia. Contact David Lee, d.lee_bluesat@lycos.com. Undergraduate microsat project, due for launch in 2004.

13. OMNI Project, GSFC Code 588, CANDOS attached shuttle payload on STS-107, July 2002. Point of contact is James Rash. Uses a pc-104 form factor 80686 cpu, with RedHat Linux 6.1, and a custom HDLC device driver, written in-house.

4. **Partnering and collaborating works**.

The work was conducted by a team from QSS Group, Inc., in conjunction with NASA/GSFC Codes 586, 582, and 588. The government-industry team approach worked smoothly, and we cooperating and collaborated with various projects, most notably, the OMNI Project in Code 588. By the nature of the mutual interests, code-sharing within the spirit of the gnu public license was possible. Resource sharing also was commonplace. The OMNI Project hosted the UoSat-12 breadboard that we used for development and test. It is possible for a government-industry team to work together successfully towards common goals, with diminished parochialism and territorial behavior. If we started this project today with our current knowledge, we would choose to work with the same partners.

The spirit of the Free Software movement is based on shared development and debugging This approach has been shown to work in many cases (Linux, the OpenOffice Suite, Mozilla Browser, the gnu tools). We were able to find many of the software tools and elements we needed in the Internet community and at GSFC, that we then used to build on. Examples of these include the IP-in-Space Security approach, the HDLC driver (Omni Project), device drivers for the CAN and 1553 buses, the RAID and ramdisk device drivers, and the library of Posix-compliant flight application software by Code 582. Most importantly, we were able to discuss issues synergistically with other Linux developers at GSFC and throughout the world.

5. **An approach is needed to allow and manage Open Source software development for Mission Critical NASA Projects**.

The FlightLinux Project explored new issues in the use of "free software" and open-source code, in a mission critical application. Open-source code, as an alternative to proprietary software has advantages and disadvantages. The chief advantage is the availability of the source code, with which a competent programming team can develop and debug applications, even those with tricky timing relationships. The Open-Source code available today for Linux supports international and ad hoc standards. The use of a standards-based architecture has been shown to facilitate functional integration. It is a misconception that "free software" is necessarily available for little or no cost. The "free" part refers to the freedom to modify the source code.

A disadvantage of developing with Open Source may be the perception that freely downloadable source code might not be mature or trustworthy. Countering this argument is the growing experience that the Open-Source offerings are as good as, and sometimes better than the equivalent commercial products. What is needed, however, is a strong configuration control mechanism. For the FlightLinux product, the FlightLinux Team assumed the responsibility of making and maintaining the "official" version.

Many issues on the development and use of Open-Source software on government-funded and mission-critical applications are still to be explored. Debates continue in the media about the relative merits of Open Source versus proprietary software with respect to security issues. This is an area that is both timely and critical, and will need to result in a definition of policy and preferred approach, as well as guidelines to projects.

References

1. HTTP://FlightLinux.gsfc.nasa.gov/docs/Target_Arch_Report.pdf

In the *Target Architecture Technical Report*, we examine the current, near term, and projected computer architectures that will be used onboard spacecraft. From this list, we examine the feasibility and availability of Linux. The choice of the actual architecture for implementation will be determined more by opportunity of a flight than by choice of the easiest or most optimum architecture.

2. HTTP://FlightLinux.gsfc.nasa.gov/docs/POSIX.pdf

The *POSIX Report* examines and documents the POSIX-compliant aspects of Linux and other Flight Operating systems as well as the POSIX-compliant nature of legacy flight application software. This is an ongoing effort by GSFC Code 582, the Flight Software Branch.

3. HTTP://FlightLinux.gsfc.nasa.gov/docs/onboard_lan.pdf

The *Onboard LAN Architecture Report* discusses the physical-level interfaces on existing and emerging missions as well as the device drivers required to support Internet Protocol (IP) over these interfaces. The CCSDS committee and the OMNI Project (GSFC, Code 588) are doing ongoing work in this area. The choice of a demonstration flight defines which interfaces will need to be implemented first. In addition, those interfaces with COTS drivers and those for which device drivers need to be defined are delineated.

4. HTTP://FlightLinux.gsfc.nasa.gov/docs/ Bulk_Memory_Device_Driver.pdf

The Bulk Memory Device Driver Report defines the approach to be taken to implement the Linux file system in the bulk memory ("tape recorder") of the spacecraft onboard computer. It defines which elements are COTS and which need to be developed.

5. HTTP://FlightLinux.gsfc.nasa.gov/docs//Embedded_testbed.pdf

The Embedded Test Bed Report defines the requirements and architecture for the facility to develop and validate the operating system code for the flight experiment. Guidance has been drawn from similar past facilities.

6. Linux usage on the Space Shuttle:
http://www.faho.rwth-aachen.de/%7Ematthi/linux/LinuxInSpace.html

7. Linux usage on the International Space Station Freedom
http://www.sheflug.co.uk/featuresoft.htm
http://www.linuxjournal.com/article.php?sid=3024

8. http://www.newsforge.com/article.pl?sid=01/03/13/2112221

9. Sandred, Jon, "Managing Open Source Projects, " NY: John Wiley & Sons, Inc. 2001.

10. Raymond, Eric, "The Cathedral and the Bazaar," O'Reilly & Associates, 1999.
also, http://tuxedo.org/~esr/writings/cathedral-bazaar/

11. Autonomy Technology and Onboard Processing Study, Space Technology 7 - Autonomy and Autonomy Pilot Project, NASA/GSFC, 8 February 2002.